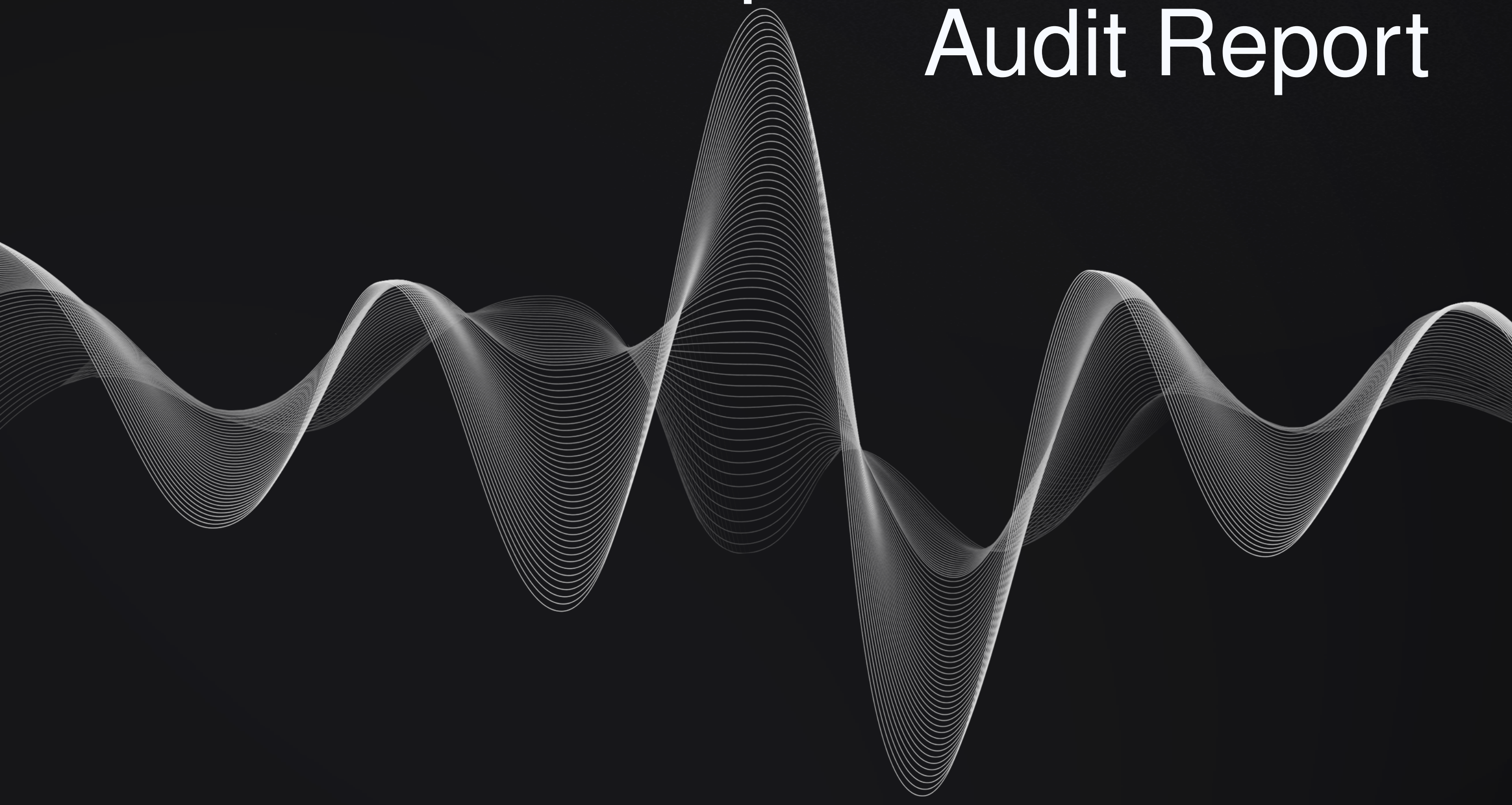
RESONANCE

Talisman

Talisman Wallet

Talisman Staking And Airdrop Smart Contract Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.1)

Audit_Report_TLMN-STA_FINAL_21

Sep 16, 2025	○	v0.1	Luis Arroyo: Initial draft
Sep 16, 2025	○	v0.2	Luis Arroyo: Added findings
Sep 16, 2025	🟢	v1.0	Charles Dray: Approved
Sep 22, 2025	○	v1.1	Luis Arroyo: Reviewed findings
Sep 22, 2025	○	v1.2	Luis Arroyo: Added findings
Sep 23, 2025	○	v1.3	Luis Arroyo: Reviewed findings
Sep 24, 2025	🟡	v2.0	Charles Dray: Finalized
Oct 7, 2025	○	v2.1	Charles Dray: Published

Points of Contact	Toby	Talisman	t@talisman.xyz
	Charles Dray	Resonance	charles@resonance.security
Testing Team	Luis Arroyo	Resonance	luis.arroyo@resonance.security
	Ilan Abitbol	Resonance	ilan@resonance.security
	João Simões	Resonance	joao@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Talisman contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between September 12, 2025 and September 16, 2025. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 3 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Talisman with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

The protocol contains a staking contract, which has pre-funded rewards. The rewards are based on elapsed time and calculated in rewardPerToken. Users can stake, claim, request/complete/cancel withdrawals (which are delayed) and propose new reward rates or withdraw delay.

The airdrop smart contract lets eligible addresses claim ERC20 tokens using a Merkle proof, blocking double-claims and supporting an airdrop "closed" switch. The amounts below threshold transfer instantly but larger amounts are locked via Sablier Lockup over 52 weeks, storing each user's streamId.

Repository Coverage and Quality

Code	Tests	Documentation
9 / 10	10 / 10	8 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is excellent**.
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 100%. Overall, **tests coverage and quality is excellent**.

- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is good**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `TalismanSociety/airdrop-contract/contracts`
- Hash: 491d08ca808c3e6607006d17d6f50f13aeed0a07
- Repository: `TalismanSociety/seek-staking/src`
- Hash: dab2000e5e12e54822bd09cd61c78e6b7de5f705

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak **-** Inconsequential or minimal damage or loss
2. Medium **-** Temporary or partial damage or loss
3. Strong **-** Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely **-** Requires substantial knowledge or effort or uncontrollable conditions
2. Likely **-** Requires technical knowledge or no special conditions
3. Very Likely **-** Requires trivial knowledge or effort or no conditions

			Likelihood	
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- ▁▂▁▂▁▂ **"Quick Win"** Requires little work for a high impact on risk reduction.
- ▂▃▅▃▂ **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- ▃▅█▅▃ **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	🟪	Reserve Grief Via Rounding In _updateGlobal()	▂▃▅▃▂	Resolved
RES-02	🟪	Reserve Grief Via Emergency Exit In _updateGlobal()	▂▃▅▃▂	Resolved
RES-03	🟥	API Key Exposed In Tests	▂▃▅▃▂	Resolved
RES-04	🟥	Denial of Service Due To Integer Overflow	▂▃▅▃▂	Resolved
RES-05	🟧	Unsafe Downcast	▁▂▁▂▁▂	Resolved
RES-06	🟩	Missing Min/Max Limits In vestTreshold And rewardRate	▂▃▅▃▂	Resolved
RES-07	🟩	Missing Zero Address Validation On constructor()	▂▃▅▃▂	Resolved
RES-08	⬜	Missing 0-then-set Pattern On approve()	▂▃▅▃▂	Resolved
RES-09	⬜	Floating Pragma	▁▂▁▂▁▂	Resolved

Reserve Grief Via Rounding In _updateGlobal()

Critical **RES-TLMN-STA01** Data Validation **Resolved**

Code Section

- src/SinglePoolStaking.sol#L681

Description

The `_updateGlobal()` function computes `rewardPerTokenStored = (newly * 1e18) / totalStaked` and then unconditionally decrements `rewardReserves` by `newly`. When `rewardPerTokenStored` is 0, no user accrues rewards but `rewardReserves` still decreases by newly. In this case, tokens are removed from `rewardReserves` without being allocated to users. Those tokens remain in the contract's balance but are unclaimable and cannot be rescued (as `rescueTokens()` disallows the reward token), effectively becoming stuck.

This scenario can be achieved following these steps.

- Set `rewardRate` = 1 Wei/sec (this is allowed as min is not constrained).
- If `totalStaked > 1 ether`, after 1 second, `newly = 1` which means `rewardPerTokenStored = 1 * 1e18 / totalStaked = 0.`
- Each `_updateGlobal()` call with a 1-second elapsed will consume 1 Eei from `rewardReserves` while `rewardPerTokenStored` stays unchanged.

Recommendation

It is recommended to check the `rewardPerTokenStored` before decrementing `rewardReserves` in order to avoid loss of funds.

Status

The issue has been fixed in db01f56a71866b38b04f54cd67a9c3e5fe9b8e32.

Reserve Grief Via Emergency Exit In _updateGlobal()

Critical **RES-TLMN-STA02** Data Validation **Resolved**

Code Section

- `src/SinglePoolStaking.sol#L681`

Description

When calling `emergencyWithdrawal()` function, `_updateGlobal()` is called first, which consumes reserves for the elapsed window even though the exiter then forfeits all rewards. If the exiter is the only staker, that consumed portion becomes unclaimable by anyone, effectively blocking reserves.

An attacker can repeatedly stake a small amount, wait, and then call `emergencyWithdraw()` to deplete `rewardReserves` without ever claiming rewards.

This could be a drain vector once `emergencyExitEnabled` is set to `true`.

Recommendation

It is recommended to update the user's forfeited amount and add it back to `rewardReserves` before setting to 0 the user state.

Status

The issue has been fixed in 2555f92613e3d89875b651613f86857b3eef247f.

API Key Exposed In Tests

High **RES-TLMN-STA03** Sensitive Data Exposure **Resolved**

Code Section

- `contracts/test/MerkleAirdrop.t.sol#L40`

Description

API keys, bearer tokens, and secrets are authentication credentials that allow users (or systems) to access protected endpoints The alchemy API key is being used in the test file provided.

Recommendation

It is recommended to remove the API key from tests and set it in a `.env` file.

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd.

Denial of Service Due To Integer Overflow

High **RES-TLMN-STA04** Denial of Service **Resolved**

Code Section

- `contracts/src/MerkleAirdrop.sol#L98`

Description

It is possible that the owner may not be able to recover the tokens due to improper accounting. Due to the vulnerability mentioned in RES-04, it is possible that allowance may be greater than balance, thus creating a denial scenario when collecting tokens. Below are the steps to reach this scenario.

- User1 claims an `amount > type(uint128).max` which uses the vesting path and the contract raises `allowance(address(this), lockup)`.
- The unsafe downcast mentioned in `RES-04` allows a stream with an incorrect amount and let the airdrop with more allowance than expected.
- User2 claims and uses the transfer path. The balance is correct as the `createWithDurationsLL()` only transferred the overflow amount
- owner tries to call `withdrawRemaining()` but as the allowance is higher than expected, the transaction will revert.

Recommendation

It is recommended to use safer casting, align approval with pull amount from streams and avoid using allowance as accounting. in this case, after `createWithDurationsLL()` call, the allowance should be consumed to zero by `transferFrom`, leaving no stale approval.

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd.

Unsafe Downcast

Medium **RES-TLMN-STA05** Data Validation **Resolved**

Code Section

- `contracts/src/MerkleAirdrop.sol#L79`

Description

When users claim the airdrop, there is a downcast triggered when the amount is equal or greater than the threshold. in this case, a new stream is created with some parameters in it. One of this parameters is the claimed amount but it is truncated to `uint128`, which may lead to errors in the protocol, as the airdrop approves the `uint256` value.

Recommendation

It is recommended to make use of a safe-cast library. E.g. OpenZeppelin's SafeCast .

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd.

Missing Min/Max Limits In vestTreshold And rewardRate

Low

RES-TLMN-STA06 Data Validation **Resolved**

Code Section

- `contracts/src/MerkleAirdrop.sol#L41`
- `src/SinglePoolStaking.sol#L313`

Description

There is no upper or lower limits when setting `vestTreshold` or `rewardRate` in constructors. This may cause a bad user experience and protocol usage if they are not set between reasonable values.

Recommendation

It is recommended to limit the values that privileged roles may set for the mentioned parameters.

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd.

Missing Zero Address Validation On constructor()

Low

RES-TLMN-STA07 Data Validation **Resolved**

Code Section

- `contracts/src/MerkleAirdrop.sol#L37`

Description

Throughout the protocol there are multiple instances where input parameters are not being validated against the Zero Address, most of which are used to perform external calls, allowing for undefined behavior within the protocol.

It should be noted that although this occurs mostly in the constructor, mistakes can be made by the deployer of the smart contracts, allowing for unwanted transactions to take place in the future.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure proper variable values and external calls are handled properly and successfully.

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd.

Missing 0-then-set Pattern On approve()

Info **RES-TLMN-STA08** Data Validation **Resolved**

Code Section

- `contracts/src/MerkleAirdrop.sol#L64`

Description

"Zero-then-set" means resetting a spender's allowance to 0 before setting it to a new non-zero value. It's used to avoid two issues: the classic ERC-20 approve frontrun and non-standard tokens (e.g., USDT) that reject changing a non-zero allowance directly to another non-zero value.

Recommendation

It is recommended to follow the "Zero-then-set" pattern. Using `increaseAllowance` is not recommended as OpenZeppelin removed it from their core ERC20 in v5.x.

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd.

Floating Pragma

Info **RES-TLMN-STA09** Code Quality **Resolved**

Code Section

- `contracts/src/MerkleAirdrop.sol#L2`
- `src/SinglePoolStaking.sol#L2`

Description

The project uses floating pragmas `^0.8.24` and `^0.8.20`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Recommendation

It is recommended to use a strict and locked pragma version for solidity code. Preferably, the version should be neither too new or too old.

Status

The issue has been fixed in 20a1f2e5b13152744cf53a52bddd824521e5dacd and db01f56a71866b38b04f54cd67a9c3e5fe9b8e32.

Proof of Concepts

RES-01 Reserve Grief Via Rounding In _updateGlobal()

```
function testReserveDrain_ViaRounding() public {
    _stake(alice, 100 ether);

    staking.proposeRewardRate(1);
    vm.warp(block.timestamp + 1);
    staking.executeRewardRateChange();
    assertEq(staking.rewardRate(), 1);

    uint256 reservesBefore = staking.rewardReserves();

    console.log("-->", staking.rewardReserves());

    uint256 iterations = 20;
    vm.startPrank(alice);
    for (uint256 i = 0; i < iterations; i++) {
        vm.warp(block.timestamp + 1); // elapsed = 1s → newly = 1
        staking.requestWithdrawal(1);
        staking.cancelWithdrawal();
    }
    vm.stopPrank();

    console.log("-->", staking.rewardReserves());

    uint256 reservesAfter = staking.rewardReserves();

    assertEq(reservesBefore, reservesAfter + iterations);
}
```

RES-02 Reserve Grief Via Emergency Exit In _updateGlobal()

```
function testBlockOnEmergencyExit() public {
    uint256 elapsed = 10;

    staking.setEmergencyExitEnabled(true);
    _stake(alice, 100 ether);

    vm.warp(block.timestamp + elapsed);
    uint256 resBefore = staking.rewardReserves();
    uint256 rate = staking.rewardRate();
    uint256 expectedConsume = elapsed * rate;

    vm.prank(alice);
    staking.emergencyWithdraw();

    uint256 resAfter = staking.rewardReserves();
    assertEq(resBefore - resAfter, expectedConsume, "reserves not consumed as
↪   expected");
```

```
    assertEq(staking.balanceOf(alice), 0, "alice stake not cleared");
    assertEq(staking.earned(alice), 0, "alice rewards not forfeited");
}
```

RES-04 Denial of Service Due To Integer Overflow

```
function testUsersClaimThenWithdraw() public {
    uint256 amount = type(uint128).max;
    TestToken token1 = new TestToken();
    token1.mint(owner, amount);
    token1.mint(owner, 100);

    MerkleHelper helper1 = new MerkleHelper();
    bytes32[] memory leaves1 = new bytes32[](2);
    leaves1[0] = keccak256(abi.encodePacked(user1, amount1));
    leaves1[1] = keccak256(abi.encodePacked(user3, amount + 10));

        // Compute root
    bytes32 root = helper.computeRoot(leaves1);
    MerkleAirdrop airdrop1 = new MerkleAirdrop(address(token1), root,
 ↪  address(sablierLockup), vestingTreshold);
    token1.transfer(address(airdrop1), 100 + amount);

    bytes32[] memory proof1 = helper1.getProof(leaves1, 0);
    bytes32[] memory proof3 = helper1.getProof(leaves1, 1);

    vm.prank(user3);
    airdrop1.claim(amount + 10, proof3);

    vm.prank(user1);
    airdrop1.claim(amount1, proof1);

    vm.expectRevert();
    airdrop1.withdrawRemaining();
}
```